Compositional Verification of Procedural Programs using Horn Clauses over Integers and Arrays

Anvesh Komuravelli
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA, USA

Nikolaj Bjørner
Microsoft Research
Redmond, WA, USA

Arie Gurfinkel
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA, USA

Kenneth L. McMillan
Microsoft Research
Redmond, WA, USA

***Abstract*—We present a compositional SMT-based algorithm for safety of procedural C programs that takes the heap into consideration as well. Existing SMT-based approaches are either largely restricted to handling linear arithmetic operations and properties, or are non-compositional. We use Constrained Horn Clauses (CHCs) to represent the verification conditions where the memory operations are modeled using the extensional theory of arrays (ARR). First, we describe an exponential time quantifier elimination (QE) algorithm for ARR which can introduce new quantifiers of the index and value sorts. Second, we adapt the QE algorithm to efficiently obtain under-approximations using models, resulting in a polynomial time Model Based Projection (MBP) algorithm. Third, we integrate the MBP algorithm into the framework of compositional reasoning of procedural programs using may and must summaries recently proposed by us. Our solutions to the CHCs are currently restricted to quantifier-free formulas. Finally, we describe our practical experience over SV-COMP'15 benchmarks using an implementation in the tool SPACER.**

I. INTRODUCTION

While many existing algorithms for SMT-based model checking of sequential programs are limited to handling program operations modeled using Linear Arithmetic (reals and integers) [], verification of real-world programs requires us to consider the heap as well. When the input program does not have recursive procedures, one can try eliminating the memory operations by inlining all procedure calls and performing compiler optimizations to lower memory into registers (e.g., [3], [13]). However, this approach has several drawbacks, such as (a) the inlined program can be exponentially larger than the original program making it harder to verify, (b) the resulting safety proofs are non-compositional, (c) it is hard to trace the resulting counterexamples back to the original program, and (d) even with inlining, it is not always possible to lower memory into registers without a significant blowup in program size. Instead, we consider program representations that faithfully model the heap using the extensional theory of arrays (ARR) [12]. In this paper, we present an efficient, compositional, and SMT-based algorithm for safety of procedural C programs modeled using the theories of Linear Integer Arithmetic (LIA) and ARR. We restrict ourselves to discovering safety proofs involving only quantifier-free assertions.

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. This material has been approved for public release and unlimited distribution. DM-0002442.

SMT-based model checking is traditionally based on monolithic *Bounded Model Checking* (BMC) [5] which iteratively checks satisfiability of formulas that symbolically represent various unwindings of the program. However, the size of these formulas can grow exponentially in the input size of the program and we recently developed a compositional framework for SMT-based model checking to address this [16]. To check safety of an input program, the compositional approach iteratively creates and checks local reachability queries for individual procedures. To avoid redundancy and enable reuse, we maintain two kinds of summaries for each procedure – one for under-approximating and the other for over-approximating its behavior. The creation of new reachability queries and summaries involves existentially quantifying auxiliary variables. To efficiently eliminate the quantifiers, we also proposed a technique called *Model Based Projection* (MBP) that obtains under-approximations of the quantified formulas. This was quite effective in practice for Linear Arithmetic.

In this paper, we extend the above compositional framework to ARR, by developing an MBP procedure for ARR. We first show that, given a quantified formula $\exists a \cdot \varphi(a, \overline{x})$, where a is an array variable and φ is quantifier-free, there exists an equivalent formula $\exists \overline{i}, \overline{v} \cdot \psi(\overline{i}, \overline{v}, \overline{x})$, where $\overline{i}$ and $\overline{v}$ are variables of the index-sort and the value-sort of a, respectively (Sec. IV-A). We will then show how to adapt this quantifier elimination procedure to obtain an MBP procedure that efficiently under-approximates existentially quantified formulas over ARR (Sec. IV-B).

We describe how the MBP procedure for ARR is integrated with MBP procedures for Linear Arithmetic and with the rest of the framework to obtain a new compositional algorithm (Sec. V). In order to obtain over-approximations of procedure behaviors, the framework also utilizes Craig Interpolation []. Using MBP to efficiently eliminate auxiliary variables results in a straightforward interpolation procedure for ARR that infers the weakest interpolants. However, it is interesting to explore other interpolation procedures which can help expand the class of programs that can be handled by our approach.

While we target C programs for verification, our algorithm and implementation are at the level of Constrained Horn Clause (CHC) fragment of First Order Logic (FOL) which is used to encode the verification conditions. This makes the framework very flexible and directly applicable to other

problems that are reducible to CHC [12].

The algorithm we propose is implemented in our tool SPACER [] using SEAHORN [] to encode the C programs logically into the HornSMT format of CHCs. Using SPACER, we evaluate the algorithm experimentally using SV-COMP 2015 benchmarks (Sec. VI).

II. PRELIMINARIES

We consider a first-order language with equality whose signature $\mathcal{S}$ contains basic sorts (e.g., `bool` of Booleans, `int` of integers, etc.) and array sorts. An array sort $\texttt{arr}(I, V)$ is parameterized by a sort of indices I and a sort of values V. We assume that I is always a basic sort. For every array sort $\texttt{arr}(I, V)$, the language has the usual function symbols $rd : \texttt{arr}(I, V) \times I \to V$ and $wr : \texttt{arr}(I, V) \times I \times V \to \texttt{arr}(I, V)$ for reading from and writing to the array. Intuitively, $rd(a, i)$ denotes the value stored in the array a at the index i and $wr(a, i, v)$ denotes the array obtained from a by replacing the value at the index i by v. We use an axiomatization of these function symbols using the extensional theory of arrays (ARR). In particular, we have the following two axiom schema:

Read-after-write

$\forall a : \texttt{arr}(I, V)\ \forall i, j : I\ \forall v : V$

$$\begin{aligned}(i = j &\implies rd(wr(a, i, v), j) = v) \wedge \\ (i \neq j &\implies rd(wr(a, i, v), j) = rd(a, j))\end{aligned}$$

Extensionality

$\forall a, b : \texttt{arr}(I, V) \cdot (\forall i : I \cdot rd(a, i) = rd(b, i)) \implies a = b$

Intuitively, the first schema says that after modifying an array a at index i, a read results in the new value at index i and $rd(a, j)$ at every other index j. The second schema says that if two arrays agree on the values at every index location, the arrays are equal. Let $\bar{i} : I$ and $\overline{v} : V$ be vectors of index and value terms of the same length m. We write $wr(a, \bar{i}, \overline{v})$ to denote $wr(wr(\dots wr(a, \bar{i}(0), \overline{v}(0)) \dots), \bar{i}(m), \overline{v}(m))$, where $\bar{i}(k)$ denotes the kth element of $\bar{i}$. For an index term $j : I$, we write $j \in \bar{i}$ to denote $\bigvee_{k=1}^{m}(j = \bar{i}(k))$. Unless specified otherwise, $\mathcal{S}$ contains no other function symbols.

For arrays a and b of sort $\texttt{arr}(I, V)$, and a (possibly empty) vector of index terms $\bar{i}$, we write $a =_{\bar{i}} b$ to denote $\forall j : I \cdot \big(j \notin \bar{i} \implies rd(a, j) = rd(b, j)\big)$ and call such formulas *partial equalities* [18]. Using extensionality, one can easily show the following

$$a =_{\emptyset} b \equiv a = b \tag{1}$$

$$wr(a, j, v) =_{\bar{i}} b \equiv \begin{array}{l}\big(j \in \bar{i} \wedge a =_{\bar{i}} b\big) \vee \\ \Big(j \notin \bar{i} \wedge a =_{\bar{i}, j} b \wedge rd(b, j) = v\Big)\end{array} \tag{2}$$

$$a =_{\bar{i}} b \equiv \exists \overline{v} : V \cdot a = wr(b, \bar{i}, \overline{v}) \tag{3}$$

Let φ be a formula with free variables $\overline{x}$ (in some fixed order). We sometimes write it as $\varphi(\overline{x})$. We write $\varphi[t]$ to denote that a term or an atom t appears in φ. We assume the usual definition of satisfiability modulo theories (SMT).

Safety of procedural programs can be reduced to SMT of a set of a special kind of formulas known as *Constrained Horn Clauses* (CHCs) [6], [16], [12]. A *Constrained Horn Clause* (CHC) is a formula of the form

$$\forall \overline{x} \cdot \underbrace{\bigwedge_{k=1}^{m} P_k(\overline{x}_k) \wedge \varphi(\overline{x})}_{body} \implies head$$

where P_k's are predicate symbols not present in $\mathcal{S}$, $\overline{x}_k \subseteq \overline{x}$ and $|\overline{x}_k|$ is equal to the arity of P_k for every k, φ is a formula over $\mathcal{S}$, and $head$ is either an application of a fresh predicate symbol or another formula over $\mathcal{S}$. In this work, we are interested in *quantifier-free, first-order interpretations* of the P_k's that satisfy the given set of CHCs. We use *body* to refer to the antecedent of the CHC, as shown above. A CHC is called a *query* if $head$ is a formula over $\mathcal{S}$ and otherwise, it is called a *rule*. If $m \leq 1$ in the body, the CHC is *linear* and is *non-linear* otherwise. Following the convention of logic programming literature, we represent the above CHC as $head \leftarrow P_1(\overline{x}_1), \dots, P_m(\overline{x}_m), \varphi(\overline{x})$.

Intuitively, the body of each procedure can be encoded as a rule CHC, where the fresh predicate symbols denote *over-approximating summaries* of procedures. Safety assertions can be encoded as query CHCs. Any given set of CHCs can be transformed to an equisatisfiable set of three CHCs with a single predicate symbol of the following canonical form (assuming that $\mathcal{S}$ has a sort with at least two elements, e.g., `bool`; see Appendix for details).

$$\begin{aligned}Inv(\overline{x}) &\leftarrow init(\overline{x}) \qquad \neg bad(\overline{x}) \leftarrow Inv(\overline{x}) \\ Inv(\overline{x}') &\leftarrow Inv(\overline{x}), Inv(\overline{x}^o), tr(\overline{x}, \overline{x}^o, \overline{x}')\end{aligned} \tag{4}$$

Intuitively, Inv denotes an inductive invariant, $\overline{x}$ denotes the state-variables, $\overline{x}'$ denotes their next-state values, and $\overline{x}^o$ denotes the potential parameters of a procedure call. Note that, if tr is independent of $\overline{x}^o$, $Inv(\overline{x}^o)$ can be effectively dropped from the last CHC and Inv denotes an inductive invariant of a traditional transistion system. For a formula $\varphi(\overline{x})$, we write φ' and φ^o to denote $\varphi[\overline{x}'/\overline{x}]$ and $\varphi[\overline{x}^o/\overline{x}]$, respectively.

In the sequel, we restrict to this canonical form and assume that $init$, tr, and bad are quantifier-free. We define a state transformer $\mathcal{F}(\varphi_A(\overline{x}), \varphi_B(\overline{x}))$ as $(\varphi_A(\overline{x}) \wedge \varphi_B(\overline{x}^o) \wedge tr(\overline{x}, \overline{x}^o, \overline{x}')) \vee init(\overline{x}')$ and abuse the notation to write $\mathcal{F}(\varphi_A)$ for $\mathcal{F}(\varphi_A, \varphi_A)$.

Consider an existentially quantified formula $\varphi(\overline{y}) = \exists \overline{x} \cdot \varphi_m(\overline{x}, \overline{y})$ where φ_m is quantifier-free. A function $Proj_{\varphi}$ from models of φ_m to quantifier-free formulas over $\overline{y}$ is called a *Model Based Projection* (MBP) [16] iff (a) $Proj_{\varphi}$ has a finite image, (b) $\varphi \equiv \bigvee_{M \models \varphi_m} Proj_{\varphi}(M)$, and (c) for every $M \models \varphi_m$, it also holds that $M \models Proj_{\varphi}(M)$. In an earlier work, we developed efficient MBP functions for the theories of Linear Real Arithmetic and Linear Integer Arithmetic [16].

Given formulas $\varphi_A(\overline{x}, \overline{z})$ and $\varphi_B(\overline{y}, \overline{z})$ with $\overline{x} \cap \overline{y} = \emptyset$ and $\varphi_A \implies \varphi_B$, a Craig Interpolant [8], denoted $\text{ITP}(\varphi_A, \varphi_B)$, is a formula $\varphi_I(\overline{z})$ such that $\varphi_A \implies \varphi_I$ and $\varphi_I \implies \varphi_B$.

III. The Compositional Verification Framework

Our approach for checking satisfiability of the CHCs in (4) is based on the SPACER framework for SMT-based model checking [16]. Compared to other SMT-based algorithms (e.g., [4], [11], [14], [17]), the key distinguishing feature of SPACER is compositional reasoning. That is, instead of checking satisfiability of monolithic SMT formulas for various program unwindings, SPACER iteratively creates and checks local reachability queries for individual procedures. At a high level, such a local reasoning is similar to IC3 [7], [10], a SAT-based algorithm for safety of finite-state transition systems, and GPDR [10], its extension to Linear Real Arithmetic. Similar to existing SMT-based algorithms, including IC3 and GPDR, SPACER maintains a sequence of over-approximations of procedure behaviors, called *may summaries*, corresponding to various program unwindings. However, unlike other approaches, SPACER also maintains under-approximations of procedure behaviors, called *must summaries*, to avoid redundant reachability queries. Another distinguishing feature of SPACER is the use of MBP for efficiently handling existentially quantified formulas to create a new query or a must summary. Alg. 1 gives a simplified description of SPACER in the context of the CHCs in (4) using a set of rules that can be applied non-deterministically. We will briefly describe the rules below and then mention some implementation aspects.

Input: Formulas $init(\overline{x}), tr(\overline{x}, \overline{x}^o, \overline{x}'), bad(\overline{x})$
Output: Inductive invariant (FO interpretation of Inv satisfying (4)) or UNSAFE

if $(init \wedge bad)$ *satisfiable* **then return** UNSAFE
// initialize data structures
$Q := \emptyset$ *// set of pairs* $\langle \varphi, i \rangle, i \in \mathbb{N}$
$N := 0$ *// max level, or recursion depth*
$\mathcal{O}_0 = init, \mathcal{O}_i = \top, \forall i > 0$ *// may summary sequence*
$\mathcal{U} = init$ *// must summary*
forever *non-deterministically* **do**
- **(Candidate) [** $(\mathcal{O}_N \wedge bad)$ *satisfiable* **]**
  $Q := Q \cup \langle \varphi, N \rangle$, for some $\varphi \implies \mathcal{O}_N \wedge bad$
- **(DecideMust) [** $\langle \varphi, i+1 \rangle \in Q,\ M \models \mathcal{F}(\mathcal{O}_i, \mathcal{U}) \wedge \varphi'$ **]**
  $Q := Q \cup \langle \text{MBP}(\exists \overline{x}^o, \overline{x}' \cdot \mathcal{F}(\mathcal{O}_i, \mathcal{U}) \wedge \varphi', M), i \rangle$
- **(DecideMay) [** $(\varphi, i+1) \in Q,\ M \models \mathcal{F}(\mathcal{O}_i) \wedge \varphi'$ **]**
  $Q := Q \cup \langle \text{MBP}(\exists \overline{x}, \overline{x}' \cdot \mathcal{F}(\mathcal{O}_i) \wedge \varphi', M)[\overline{x}/\overline{x}^o], i \rangle$
- **(Leaf) [** $(\varphi, i) \in Q,\ \mathcal{F}(\mathcal{O}_{i-1}) \implies \neg\varphi',\ i < N$ **]**
  $Q := Q \cup \langle \varphi, i+1 \rangle$
- **(Successor) [** $\langle \varphi, i+1 \rangle \in Q,\ M \models \mathcal{F}(\mathcal{U}) \wedge \varphi'$ **]**
  $\mathcal{U} := \mathcal{U} \vee \text{MBP}(\exists \overline{x}, \overline{x}^o \cdot \mathcal{F}(\mathcal{U}) \wedge \varphi', M)[\overline{x}/\overline{x}']$
- **(Conflict) [** $\langle \varphi, i+1 \rangle \in Q,\ \mathcal{F}(\mathcal{O}_i) \implies \neg\varphi'$ **]**
  $\mathcal{O}_j := \mathcal{O}_j \wedge \text{ITP}(\mathcal{F}(\mathcal{O}_i), \neg\varphi')[\overline{x}/\overline{x}'], \forall j \leq i+1$
- **(Induction) [** $(\varphi \vee \psi) \in \mathcal{O}_i,\ \mathcal{F}(\varphi \wedge \mathcal{O}_i) \implies \varphi'$ **]**
  $\mathcal{O}_j := \mathcal{O}_j \wedge \varphi, \forall j \leq i+1$
- **(Unfold) [** $\mathcal{O}_N \implies \neg bad$ **]** $N := N+1$
- **(Safe) [** $\mathcal{O}_{i+1} \implies \mathcal{O}_i$ **]** **return** invariant $\mathcal{O}_i$
- **(Unsafe) [** $(\mathcal{U} \wedge bad)$ *satisfiable* **]** **return** UNSAFE

Algorithm 1: Rule-based description of SPACER.

As shown in Alg. 1, SPACER maintains a set of reachability queries Q, a sequence of may summaries $\{\mathcal{O}_i\}_{i \in \mathbb{N}}$, and a must summary $\mathcal{U}$. Intuitively, a query $\langle \varphi, i \rangle$ corresponds to checking if φ is reachable for recursion depth i, $\mathcal{O}_i$ over-approximates the reachable states for recursion depth i, and $\mathcal{U}$ under-approximates the reachable states. N denotes the current bound recursion depth. The sequence of may summaries and N correspond to the *trace of approximations* and the maximum *level* in IC3/PDR, resepectively. For convenience, let $\mathcal{O}_{-1}$ be $\bot$. $\text{MBP}(\varphi, M)$, for a formula $\varphi = \exists \overline{v} \cdot \varphi_m$ and model $M \models \varphi_m$, denotes the result of some MBP function associated with φ for the model M.

Alg. 1 initializes N to 0 and, $\mathcal{O}_0$ and $\mathcal{U}$ to $init$. Then, it iteratively applies one of the rules in the **forever** loop, chosen non-deterministically. Each rule is presented as a guarded command "[*grd*] *cmd*", where *cmd* can be executed only if *grd* holds. **Candidate** initiates a backward search for a counterexample beginning with a set of states in bad. The potential counterexample is expanded using either **DecideMust** or **DecideMay**. **DecideMust** *jumps over* the call $Inv(\overline{x}^o)$, in the last CHC of (4), utilizing the must summary $\mathcal{U}$. **DecideMay**, on the other hand, creates a query for the call using the may summary of its calling context. **Leaf** moves an unreachable query to a higher recursion depth. **Successor** updates $\mathcal{U}$ when a query is known to be reachable. **Conflict** updates may summaries when a query is known to be unreachable. **Induction** strengthens may summaries using induction relative to $\mathcal{O}_i$. **Unfold** increments the bound on the recursion depth. **Safe** returns $\mathcal{O}_i$ as invariant when the sequence of may summaries converges. **Unsafe** applies when the must summary intersects with bad.

One can show that $init \implies \mathcal{O}_0$ and for a fixed N and every $0 < i \leq N$, $\mathcal{F}(\mathcal{O}_{i-1}) \implies \mathcal{O}_i$, $\mathcal{O}_{i-1} \implies \mathcal{O}_i$, and $\mathcal{O}_i \implies \neg bad$. Moreover, $\mathcal{F}^i(init) \implies \mathcal{O}_i$ for all i, and $\mathcal{U} \implies \mathcal{F}^N(init)$. Thus, $\{\mathcal{O}_i\}_{i \in \mathbb{N}}$ and $\mathcal{U}$, respectively, over- and under-approximate reachable states and SPACER is sound.

In the description above, we left out many implementation details and we mention a few of them here. For efficiency, we restrict queries to cubes. For Linear Arithmetic, we use MBP functions that are linear in time and space. Q is maintained as a priority queue, processing queries of smaller recursion depths first. Additional constraints are imposed on the rules and their ordering to ensure termination for a fixed N [16]. For the rule **Unsafe**, our implementation also produces a counterexample in addition to returning UNSAFE.

A key ingredient in extending this framework to arrays is an efficient MBP function for ARR. This is the subject of the rest of the paper.

IV. QE and MBP for the Theory ARR

Consider an existentially quantified formula $\exists a : \texttt{arr}(I, V) \cdot \varphi$ where φ is quantifier-free. We first present an exponential time algorithm for QE, i.e., to obtain an equivalent formula that does not contain the array quantifier. Then, we present a polynomial time algorithm for MBP given a model $M \models \varphi$. Initially, we restrict the interpretations of I, the

$$\textsc{ElimEq}\ \frac{\exists a \cdot (a =_{\bar{i}} t \wedge \varphi)}{\exists \overline{v} \cdot \varphi[wr(t, \bar{i}, \overline{v})/a]}$$

where a does not appear in t and $\overline{v}$ denotes fresh variables

$$\textsc{ElimDiseq}\ \frac{\exists a \cdot \left(\varphi \ \wedge \ \bigwedge_{k=1}^{m} \neg(a =_{\bar{i}_k} t_k)\right)}{\exists a \cdot \varphi}$$

where $m \in \mathbb{N}$, a does not appear in any t_k, and a appears in φ only in read terms over a

$$\textsc{Ackermann}\ \frac{\exists a \cdot \left(\varphi \ \wedge \ \bigwedge_{k=1}^{m} s_k = rd(a, t_k)\right)}{\varphi \ \wedge \bigwedge_{1 \le k < \ell \le m} (t_k = t_\ell \implies s_k = s_\ell)}$$

where $m \in \mathbb{N}$ and a does not appear in φ, s_k's, or t_k's

Fig. 3: Rewriting rules for QE of arrays.

index sort, to infinite domains. Handling finite index domains requires a slight adaptation of the algorithms as described at the end of the section.

```
ArrayQE(∃a · φ)
1  φ1 ← (ElimWr*)(∃a · φ)
2  φ2 ← (CaseSplitEq*; FactorRd*)(φ1)
3  (⋁_{k=1}^{n} δ_k) ← LiftEqDiseqRd(φ2)
4  for k ∈ [1, n] do
5      ψ_k ← (ElimEq; ElimDiseq; Ackermann)(δ_k)
6  return ⋁_{k=1}^{n} ψ_k
```

Algorithm 2: QE for $\exists a \cdot \varphi$, where a is an array variable.

A. Quantifier Elimination

The goal of QE is to obtain an equivalent formula $\exists \overline{v} : V \cdot \psi$ where ψ is quantifier-free. Our algorithm is inspired by the decision procedure for the quantifier-free fragment of ARR by Stump et al. [18]. At a high level, the QE algorithm proceeds in 3 steps: (i) eliminate write terms using the read-after-write axiom scheme and partial equalities over arrays, (ii) eliminate (partial) equalities and disequalities over arrays, and (iii) eliminate read terms over arrays. Alg. 2 shows the pseudo-code for our QE algorithm ArrayQE using the rewrite rules in Fig. 1, 2, and 3 for equivalent transformations. We write $R(\varphi)$ to denote the result of application of R. We combine rules using the standard notation for regular expressions where R^* denotes the exhaustive iterative application of R.

Line 1 of ArrayQE eliminates write terms using the rewrite rules in Fig. 1. Here ElimWr denotes a rule in Fig. 1 chosen non-deterministically. ElimWrRd rewrites terms using the read-after-write axiom and ElimWrEq rewrites partial equalities using Eq. (2). PartialEq converts equalities into partial equalities using Eq. (1). TrivEq eliminates trivial partial equalities with identical arguments and Symm ensures that write terms on the r.h.s. of equalities are also eliminated.

Line 2 of ArrayQE rewrites the formula by case-splitting on partial equalities on the array quantifier a (via CaseSplitEq) followed by factoring out read terms over a by introducing new quantifiers of sort V (via FactorRd). Note that, as presented, these two rules can be applied indefinitely as the partial equalities and read terms are preserved in the conclusion of the rules. However, one can easily ensure that a given partial equality or read term is considered exactly once by apriori computing the set of all partial equalities and read terms in the formula and processing them in a sequential order. The details are straightforward and are left to the reader.

LiftEqDiseqRd on line 3 of ArrayQE performs Boolean rewriting and returns an equivalent disjunction such that in every disjunct, the partial equalities, array disequalities, and equalities over read terms appear at the beginning as conjuncts, in that order. In practice, CaseSplitEq can be implemented efficiently using an N-way case analysis for a total number of N partial equalities in the formula and this Boolean rewriting can be avoided. For each disjunct, line 5 applies the rules in Fig. 3 to eliminate the array quantifier a. ElimEq obtains a substitution term for a using the equivalence in Eq. (3). ElimDiseq is applicable when the disjunct contains no partial equalities and given that the domain of interpretation of I is infinite, one can always satisfy the disequalities and hence, they can simply be dropped. Ackermann performs the Ackermann reduction [2] to eliminate the read terms.

Note that while the rewrite rules are applicable to all array terms and equalities in the original formula, in practice, we only need to apply them to eliminate the relevant terms containing the array quantifier a. See Fig. 4 for an illustration of ArrayQE on an example.

Correctness and Complexity. We can show the following properties of ArrayQE.

Theorem *1:* ArrayQE($\exists a : \mathtt{arr}(I, V) \cdot \varphi$) returns $\exists \overline{v} : V \cdot \rho$, where ρ is quantifier-free and $\exists \overline{v} \cdot \rho \equiv \exists a \cdot \varphi$.

Proof: (*Sketch*) One can easily show that the rules in Fig. 1, 2, and 3 are equivalence preserving. The theorem follows immediately. ∎

Theorem *2:* ArrayQE($\exists a \cdot \varphi$) terminates in time exponential in the size of φ.

Proof: (*Sketch*) Line 1 of ArrayQE essentially eliminates write terms one by one and can be easily shown to terminate. Line 2 can be easily made to terminate by iterating over all partial equality and read terms. The remaining steps of the algorithm clearly terminate as well.

The complexity analysis is similar to the decision procedure by Stump et al. [18]. Let N be the size of φ. The number of disjuncts generated by any rewrite rule is bounded by N (due to the disjunction $j \in \bar{i}$ on indices in ElimWrEq). Disjunctions can be generated by the rules for every write term or partial equality and their number is bounded by N. So, the total number of disjunctions generated by the algorithm is bounded by $O(N^N)$ which is exponential in N. The size of a disjunct generated by a rule can be shown to be bounded

$$\textsc{ElimWrRd}\ \frac{\varphi[rd(wr(t,i,v),j)]}{(i=j\wedge\varphi[v])\vee(i\neq j\wedge\varphi[rd(t,j)])} \qquad \textsc{ElimWrEq}\ \frac{\varphi[wr(t_1,j,v)=_{\bar{i}}t_2]}{\begin{array}{l}(j\in\bar{i}\wedge\varphi[t_1=_{\bar{i}}t_2])\vee\\ \left(j\notin\bar{i}\wedge\varphi[t_1=_{\bar{i},j}t_2\wedge v=rd(t_2,j)]\right)\end{array}}$$

$$\textsc{PartialEq}\ \frac{\varphi[t_1=t_2]}{\varphi[t_1=_{\emptyset}t_2]}\ t_i\text{'s have array sort} \qquad \textsc{TrivEq}\ \frac{\varphi[t=_{\bar{i}}t]}{\varphi[\top]} \qquad \textsc{Symm}\ \frac{\varphi[t_1=_{\bar{i}}t_2]}{\varphi[t_2=_{\bar{i}}t_1]}\ \begin{array}{l}t_2\text{ is a write term}\\ \text{but } t_1\text{ is not}\end{array}$$

$$\textsc{ElimWr} = (\textsc{ElimWrRd} \mid \textsc{ElimWrEq} \mid \textsc{PartialEq} \mid \textsc{TrivEq} \mid \textsc{Symm})$$

Fig. 1: Rewriting rules to eliminate write terms. ElimWr denotes one of the rules chosen non-deterministically.

$$\textsc{CaseSplitEq}\ \frac{\exists a\cdot\varphi[a=_{\bar{i}}t]}{\exists a\cdot((a=_{\bar{i}}t\wedge\varphi[\top])\vee(\neg(a=_{\bar{i}}t)\wedge\varphi[\bot]))} \qquad \textsc{FactorRd}\ \frac{\exists a\cdot\varphi[rd(a,t)]}{\exists a,s\cdot(\varphi[s]\wedge s=rd(a,t))}\ \begin{array}{l}s\text{ is fresh, }t\text{ does not}\\ \text{contain array terms}\end{array}$$

Fig. 2: Rewriting rules to factor out equalities and read terms on the quantified array variable.

by a polynomial in N. CaseSplitEq can be efficiently implemented using an N-way case analysis over all equalities avoiding a Boolean rewriting on line 3 of the algorithm. Thus, the complexity of ArrayQE is exponential in N. ■

B. Model Based Projection

If a model $M \models \varphi$ is given, one can obtain an MBP by simply projecting the result of each rule application to the disjunct satisfied by M. Fig. 5 shows the modified rules corresponding to ElimWrRd, ElimWrEq, and CaseSplitEq and Fig. 6 shows the modified rule for Ackermann. Alg. 3 shows the pseudo-code for our MBP algorithm ArrayMBP. Fig. 7 illustrates ArrayMBP on the same example as in Fig. 4 for a specific choice of M as shown in the side-conditions.

ArrayMBP($\exists a\cdot\varphi, M$)
1 $\varphi_1 \leftarrow (\textsc{ProjWr}^*)(\exists a\cdot\varphi, M)$
2 $\varphi_2 \leftarrow (\textsc{ProjSplitEq}^*; \textsc{FactorRd}^*)(\varphi_1, M)$
3 $\left(\bigvee_{k=1}^{n}\delta_k\right) \leftarrow \textsc{LiftEqDiseqRd}(\varphi_2)$
4 **for** $k \in [1,n]$ **do**
5 $\quad \psi_k \leftarrow (\textsc{ElimEq}; \textsc{ElimDiseq}; \textsc{ProjAck})(\delta_k, M)$
6 **return** $\bigvee_{k=1}^{n}\psi_k$

Algorithm 3: MBP for $\exists a\cdot\varphi$, where a is an array variable, and $M \models \varphi$.

Correctness and Complexity. The size of a disjunct generated by a rule of ArrayMBP can be shown to be bound by a polynomial in the size of φ. The following is immediate.

Theorem *3:* ArrayMBP($\exists a : \texttt{arr}(I,V)\cdot\varphi, \cdot$) is an MBP and terminates in time polynomial in the size of φ.

C. Handling finite index domains

When finite interpretations of I are allowed, ElimDiseq is no longer an equivalent transformation as there may not exist an index where the arrays in the disequalities disagree on the values. However, one can use extensionality to obtain another equivalent transformation rule ElimDiseqFinite, as shown in Fig. 8. As this rule introduces new read terms over a, we need to apply FactorRd once again before Ackermann or ProjAck. Also, note that the result of QE and MBP is now of the form $\exists\bar{i} : I, \overline{v} : V\cdot\psi$.

$$\textsc{ProjWrRd}\ \frac{\varphi[rd(wr(t,i,v),j)] \qquad M\models\varphi}{\begin{cases} i=j\wedge\varphi[v] & M\models i=j\\ i\neq j\wedge\varphi[rd(t,j)] & \text{otherwise}\end{cases}}$$

$$\textsc{ProjWrEq}\ \frac{\varphi[wr(t_1,j,v)=_{\bar{i}}t_2] \qquad M\models\varphi}{\begin{cases} j=i\wedge\varphi[t_1=_{\bar{i}}t_2] & M\models j=i, i\in\bar{i}\\ j\notin\bar{i}\ \wedge\ \varphi[t_1=_{\bar{i},j}t_2\wedge v=rd(t_2,j)] & M\models j\notin\bar{i}\end{cases}}$$

$$\textsc{ProjCaseEq}\ \frac{\exists a\cdot\varphi[a=_{\bar{i}}t] \qquad M\models\varphi}{\begin{cases}\exists a\cdot(a=_{\bar{i}}t\wedge\varphi[\top]) & M\models a=_{\bar{i}}t\\ \exists a\cdot(\neg(a=_{\bar{i}}t)\wedge\varphi[\bot]) & \text{otherwise}\end{cases}}$$

$$\textsc{ProjWr} = (\textsc{ProjWrRd} \mid \textsc{ProjWrEq} \mid \textsc{PartialEq} \mid \textsc{TrivEq} \mid \textsc{Symm})$$

Fig. 5: MBP rules for write terms and equalities. ProjWr is the MBP version of ElimWr in Fig. 1.

V. CHCs over Arrays, Integers, and Booleans

In the rest of the paper, we restrict ourselves to the basic sorts of `bool` and `int`, in addition to array sorts. Furthermore, we only consider linear functions over `int`, axiomatized using Presburger Arithmetic (Linear Integer Arithmetic (LIA)) along

$$
\begin{aligned}
&\exists a \cdot (b = wr(a, i_1, v_1) \vee (rd(wr(a, i_2, v_2), i_3) > 5 \wedge rd(a, i_4) > 0)) \\
&\equiv \exists a \cdot \begin{array}{l} (\boldsymbol{i_2 = i_3} \wedge (b = wr(a, i_1, v_1) \vee (\boldsymbol{v_2 > 5} \wedge rd(a, i_4) > 0))) \vee \\ (\boldsymbol{i_2 \neq i_3} \wedge (b = wr(a, i_1, v_1) \vee (\boldsymbol{rd(a, i_3) > 5} \wedge rd(a, i_4) > 0))) \end{array} && \{\textsc{ElimWrRd}\} \\
&\equiv \exists a \cdot \begin{array}{l} (i_2 = i_3 \wedge ((\boldsymbol{a =_{i_1} b} \wedge \boldsymbol{rd(b, i_1) = v_1}) \vee (v_2 > 5 \wedge rd(a, i_4) > 0))) \vee \\ (i_2 \neq i_3 \wedge ((\boldsymbol{a =_{i_1} b} \wedge \boldsymbol{rd(b, i_1) = v_1}) \vee (rd(a, i_3) > 5 \wedge rd(a, i_4) > 0))) \end{array} && \{\textsc{PartialEq}; \textsc{ElimWrEq}\} \\
&\equiv \exists a \cdot \begin{array}{l} \left(\boldsymbol{a =_{i_1} b} \wedge \begin{array}{l} (i_2 = i_3 \wedge (rd(b, i_1) = v_1 \vee (v_2 > 5 \wedge rd(a, i_4) > 0))) \vee \\ (i_2 \neq i_3 \wedge (rd(b, i_1) = v_1 \vee (rd(a, i_3) > 5 \wedge rd(a, i_4) > 0))) \end{array}\right) \vee \\ \left(\boldsymbol{\neg(a =_{i_1} b)} \wedge \begin{array}{l} (i_2 = i_3 \wedge (v_2 > 5 \wedge rd(a, i_4) > 0)) \vee \\ (i_2 \neq i_3 \wedge (rd(a, i_3) > 5 \wedge rd(a, i_4) > 0)) \end{array}\right) \end{array} && \{\textsc{CaseSplitEq}\} \\
&\qquad \left(\left(a =_{i_1} b \wedge \underbrace{\begin{array}{l} (i_2 = i_3 \wedge (rd(b, i_1) = v_1 \vee (v_2 > 5 \wedge \boldsymbol{s_4} > 0))) \vee \\ (i_2 \neq i_3 \wedge (rd(b, i_1) = v_1 \vee (\boldsymbol{s_3 > 5} \wedge \boldsymbol{s_4} > 0))) \end{array}}_{\varphi_1}\right) \vee \right. \\
&\equiv \exists a, s_3, s_4 \cdot \left. \left(\neg(a =_{i_1} b) \wedge \underbrace{\begin{array}{l} (i_2 = i_3 \wedge (v_2 > 5 \wedge \boldsymbol{s_4} > 0)) \vee \\ (i_2 \neq i_3 \wedge (\boldsymbol{s_3 > 5} \wedge \boldsymbol{s_4} > 0)) \end{array}}_{\varphi_2}\right)\right) && \{\textsc{FactorRd}\} \\
&\qquad \wedge \boldsymbol{s_3 = rd(a, i_3)} \wedge \boldsymbol{s_4 = rd(a, i_4)} \\
&\equiv \exists a, s_3, s_4 \cdot \begin{array}{l} (\boldsymbol{a =_{i_1} b} \wedge \boldsymbol{s_3 = rd(a, i_3)} \wedge \boldsymbol{s_4 = rd(a, i_4)} \wedge \varphi_1) \vee \\ (\boldsymbol{\neg(a =_{i_1} b)} \wedge \boldsymbol{s_3 = rd(a, i_3)} \wedge \boldsymbol{s_4 = rd(a, i_4)} \wedge \varphi_2) \end{array} && \{\textsc{LiftEqDiseqRd}\} \\
&\equiv \exists v, s_3, s_4 \cdot (s_3 = rd(a, i_3) \wedge s_4 = rd(a, i_4) \wedge \varphi_1)\,[\boldsymbol{wr(b, i_1, v)/a}] \vee && \{\textsc{ElimEq}\} \\
&\quad \exists a, s_3, s_4 \cdot (s_3 = rd(a, i_3) \wedge s_4 = rd(a, i_4) \wedge \varphi_2) && \{\textsc{ElimDiseq}\} \\
&\equiv \exists v, s_3, s_4 \cdot (s_3 = rd(a, i_3) \wedge s_4 = rd(a, i_4) \wedge \varphi_1)\,[wr(b, i_1, v)/a] \vee \\
&\quad \exists s_3, s_4 \cdot (\boldsymbol{(i_3 = i_4 \implies s_3 = s_4)} \wedge \varphi_2) && \{\textsc{Ackermann}\}
\end{aligned}
$$

Fig. 4: Illustrating ArrayQE on an example.

$$
\textsc{ProjAck}\ \frac{\begin{array}{c} \exists a \cdot \left(\varphi \ \wedge \bigwedge_{k=1}^{m} s_k = rd(a, t_k)\right) \\ M \models \varphi \ \wedge \bigwedge_{k=1}^{m} s_k = rd(a, t_k) \end{array}}{\varphi \wedge \bigwedge_{1 \leq k < \ell \leq m} \begin{cases} t_k = t_\ell \wedge s_k = s_\ell & M \models t_k = t_\ell \\ t_k \neq t_\ell & \text{otherwise} \end{cases}}
$$

where $m \in \mathbb{N}$ and a does not appear in φ, s_k's, or t_k's

Fig. 6: MBP rule for ElimRd in Fig. 3.

$$
\textsc{ElimDiseqFinite}\ \frac{\exists a \cdot (\neg(a =_{\bar{i}} t) \wedge \varphi)}{\exists a, j \cdot \left(rd(a, j) \neq rd(t, j) \wedge j \notin \bar{i} \wedge \varphi\right)}
$$

where a does not appear in t

Fig. 8: Modified version of ElimDiseq for finite domains.

with a divisibility predicate. Extending the compositional framework of Spacer to this setting requires us to come up with relevant procedures for Mbp and Itp. **Conflict** is the only rule in Alg. 1 that uses Itp and $\neg\varphi'$ is an easy candidate for $\textsc{Itp}(\mathcal{F}(\mathcal{O}_i), \neg\varphi')$. Alternatively, one can use heuristics such as generalization using unsatisfiable cores or other theory-specific interpolation procedures.

We developed MBP functions for Booleans and LIA in a previous work [16]. For ARR, we described an MBP function in the previous section. When the index sort I is `int`, one can obtain a modified ProjAck for eliminating array read terms by utilizing the predicate symbol $<$ and the given model M to linearly order the index terms t_k's. This can be achieved by partitioning the set of index terms t_k's according to their interpretations in M, choosing a representative for each equivalence class, ensuring that t_k is always a representative in the equality $t_k = t_\ell$ in the rule, and linearly ordering the representatives of the various equivalence classes according to M. The resulting MBP function is linear in time and space and is more efficient (at the price of an enlarged image set).

However, the combination of arrays and integers introduces terms over the combined signature which need to be handled as well. For example, there is no equivalent quantifier-free formula for $\exists i : \texttt{int} \cdot rd(a, i) > 0$. This implies that there does not exist an MBP for the combination of LIA and ARR. In the example, the only way to under-approximate the quantification is to substitute i with its interpretation in a model $M \models rd(a, i) > 0$. Unfortunately, Spacer is no longer guaranteed to terminate even for a fixed bound on the recursion

$$
\begin{aligned}
&\exists a \cdot (b = wr(a, i_1, v_1) \vee (rd(wr(a, i_2, v_2), i_3) > 5 \wedge rd(a, i_4) > 0)) \\
&\quad \Leftarrow \exists a \cdot (\boldsymbol{i_2 \neq i_3} \wedge (b = wr(a, i_1, v_1) \vee (\boldsymbol{rd(a, i_3) > 5} \wedge rd(a, i_4) > 0))) && \{\textsc{ProjWrRd}, M \models i_2 \neq i_3\} \\
&\quad \Leftarrow \exists a \cdot (i_2 \neq i_3 \wedge ((\boldsymbol{a =_{i_1} b} \wedge \boldsymbol{rd(b, i_1) = v_1}) \vee (rd(a, i_3) > 5 \wedge rd(a, i_4) > 0))) && \{\textsc{PartialEq}; \textsc{ProjWrEq}\} \\
&\quad \Leftarrow \exists a \cdot \boldsymbol{\neg(a =_{i_1} b)} \wedge i_2 \neq i_3 \wedge (rd(a, i_3) > 5 \wedge rd(a, i_4) > 0) && \{\textsc{ProjCaseEq}, M \not\models a =_{i_1} b\} \\
&\quad \Leftarrow \exists a, s_3, s_4 \cdot \left(\neg(a =_{i_1} b) \wedge \underbrace{i_2 \neq i_3 \wedge (\boldsymbol{s_3 > 5} \wedge \boldsymbol{s_4} > 0)}_{\varphi_2} \right) && \{\textsc{FactorRd}\} \\
&\qquad\qquad \wedge \boldsymbol{s_3 = rd(a, i_3)} \wedge \boldsymbol{s_4 = rd(a, i_4)} \\
&\quad \Leftarrow \exists a, s_3, s_4 \cdot (\boldsymbol{\neg(a =_{i_1} b)} \wedge \boldsymbol{s_3 = rd(a, i_3)} \wedge \boldsymbol{s_4 = rd(a, i_4)} \wedge \varphi_2) && \{\textsc{LiftEqDiseqRd}\} \\
&\quad \Leftarrow \exists a, s_3, s_4 \cdot (s_3 = rd(a, i_3) \wedge s_4 = rd(a, i_4) \wedge \varphi_2) && \{\textsc{ElimDiseq}\} \\
&\quad \Leftarrow \exists s_3, s_4 \cdot (\boldsymbol{(i_3 = i_4} \wedge \boldsymbol{s_3 = s_4)} \wedge \varphi_2) && \{\textsc{ProjAck}, M \models i_3 = i_4\}
\end{aligned}
$$

Fig. 7: Illustrating ARRAYMBP on the example of Fig. 4 with a given model M.

depth N. Note that MBP is used in 3 rules: **DecideMay**, **DecideMust**, and **Successor**. The elimination of quantifiers in **Successor** is only an optimization and can be avoided. This is not the case with **DecideMay** or **DecideMust** without changing the structure of the queries, the considerations of which are outside the scope of this paper. In the following, we identify restrictions on the CHCs where termination is still guaranteed and for the other cases, we propose some heuristics to avoid non-termination.

There are several cases where terms over combined signatures appear in conjunction with equality terms over the index quantifier, e.g., $\exists i : \texttt{int} \cdot i = t \wedge rd(a, i) > 0$ for a term t independent of i. In these cases, the quantifier can be eliminated using equality resolution, e.g., $rd(a, t) > 0$ in the above example. Such cases seem to be natural in the case of a single procedure, i.e., when tr in (4) is independent of $\overline{x}^o$. Consider a disjunct δ in a DNF representation of tr. Now, δ represents a path in the procedure and typically, index terms (in reads and writes) in δ can be ordered such that every index term is a function of the previous index terms or the current-state variables $\overline{x}$. This makes it possible to eliminate any index variables in $\overline{x}'$ using equality resolution as mentioned above.

In general, non-termination cannot be avoided as shown by the following set of CHCs.

$$
\begin{aligned}
Inv(a, b) &\leftarrow a = b \\
\bot &\leftarrow Inv(a, b), rd(a, j) < 0, rd(b, j) > 0
\end{aligned}
$$

Here, intuitively, $Inv(a, b)$ denotes the summary of a procedure which takes a as input and produces b as output and we are interested in checking if there is sign change in the value at an index j as a result of the procedure call. For this example, **DecideMay** creates queries of the form $rd(a, k) < 0 \wedge rd(b, k) > 0$ where k is a specific integer constant. If ITP returns interpolants of the form $rd(a, k) = rd(b, k)$, it is easy to see that SPACER would not terminate even for $N = 0$.

To help alleviate the problem of non-termination, we can modify **DecideMust** and **DecideMay** as follows. Let ψ be the result of MBP in the rules, using a given model M. For every pair of array terms a, b in ψ, we strengthen ψ with the array equality $a = b$ or disequality $a \neq b$, depending on whether $M \models a = b$ holds or not. In the above example, the queries will now be of the form $rd(a, k) < 0 \wedge rd(b, k) > 0 \wedge a \neq b$. However, $rd(a, k) = rd(b, k)$ continues to be an interpolant whereas the desired interpolant is $a = b$. To reduce the dependence on specific integer constants in the learnt interpolants, and hence in the may summaries, we can further modify **Conflict** as follows. Let $\mathcal{F}(\mathcal{O}_i) \implies \neg\varphi'$ as in **Conflict**, and let $\varphi = \varphi_1 \wedge \varphi_2$ where φ_2 contains all the literals where an integer quantifier is substituted using its interpretation in a model. Using a *minimal unsatisfiable subset* (MUS) algorithm, we can generalize φ_2 to $\hat{\varphi}_2$ such that $\mathcal{F}(\mathcal{O}_i) \wedge (\varphi_1 \wedge \hat{\varphi}_2)'$ is unsatisfiable and then obtain $\text{ITP}(\mathcal{F}(\mathcal{O}_i), \neg(\varphi_1 \wedge \hat{\varphi}_2)')$. In the above example, for $i = 0$, $\mathcal{F}(\mathcal{O}_0) = (a = b)$, $\varphi_1 = (a \neq b)$, and $\varphi_2 = rd(a, k) < 0 \wedge rd(b, k) > 0$. One can show that $\hat{\varphi}_2$ is simply $\top$ and the only possible interpolant is $a = b$. In our implementation, we add such (dis-)equalities on-demand in a lazy fashion. Note that adding such (dis-)equalities to the queries is only a heuristic and may not help with termination in all cases.

VI. EXPERIMENTAL RESULTS

We have a prototype implementation of the algorithms described so far in our tool SPACER.[1] Although the description so far has focused on the canonical form of CHCs in (4), SPACER can handle arbitrary CHCs. To verify C programs, we use SEAHORN [12], which uses the LLVM infrastructure to compile the input program, optimize it, and encode the verification conditions as CHCs using the SMT-LIB2 format, which is then input to SPACER. Among other options, SEAHORN can avoid inlining procedure calls before encoding the problems as CHCs.

We evaluated SPACER using benchmarks from the software verification competition SV-COMP'15 [1]. The only other tool that is similar to SPACER is the implementation of GPDR [15] in Z3 [9], with the key differences being the use of must

[1] https://bitbucket.org/spacer/code

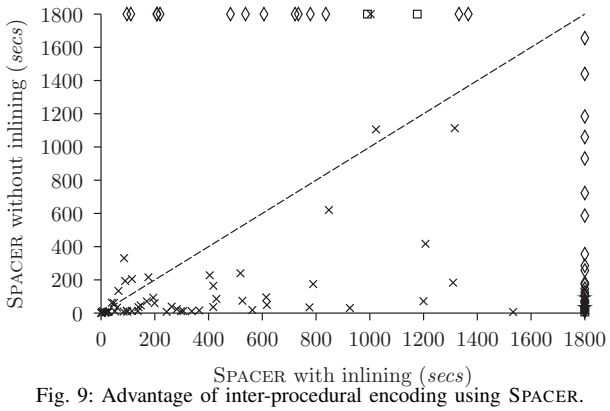

Fig. 9: Advantage of inter-procedural encoding using SPACER.

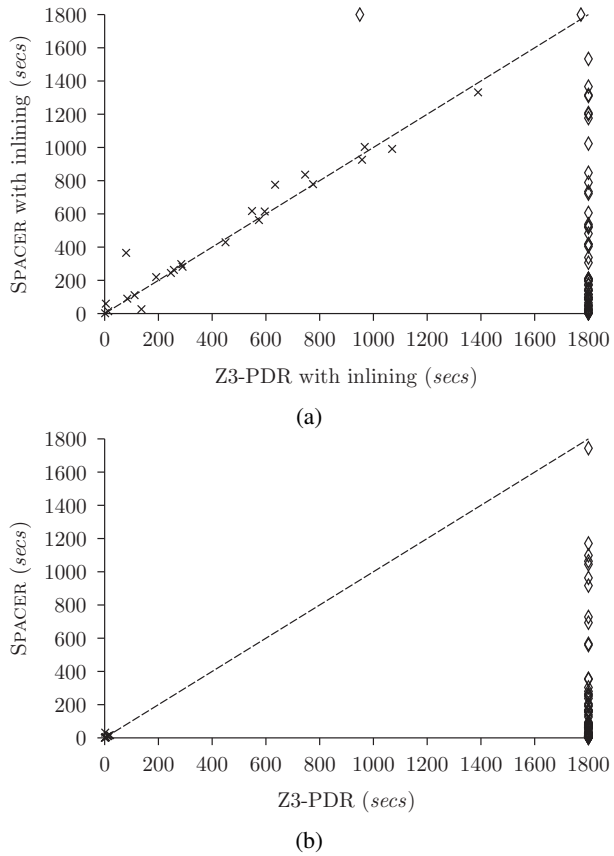

Fig. 10: SPACER vs. Z3 on hard benchmarks (a) with and (b) without inlining

summaries and MBP which are only present in SPACER. To evaluate the features in SPACER, we considered the 215 benchmarks from the *Device Drivers* category where Z3 needed more than a minute of runtime or could not be verify within the resource limits of SV-COMP [13]. All experiments have been carried out on an Ubuntu machine with a 2.2 GHz AMD Opteron(TM) Processor 6174 and 516GB RAM with resource limits of 30 minutes and 15GB for each verification task. In the scatter plots that follow, a diamond indicates a *time-out*, a star indicates a *mem-out*, and a box indicates an anomaly in the implementation.

(a) Advantage of modularity in encoding C programs. As mentioned in Sec. I, a key motivation for this work is to verify a program while preserving the procedural modularity and avoiding inlining procedure calls. While being advantageous from a usability perspective, we observed that preserving the modularity also makes verification easier. The scatter plot in Fig. 9 compares the overall time taken for the CHC encoding and SPACER's verification, when inlining in SEAHORN is turned on and off, showing an advantage when it is turned off.

(b) Advantage of our compositional framework. To see the effect of must summaries and MBP (for LIA and ARR), we compared Z3 and SPACER. The scatter plots in Fig. 10(a) and 10(b) compare the tools on the CHCs obtained when inlining in SEAHORN is turned on and off, respectively. In both cases, we clearly see that SPACER has a significant advantage. Note that, in the latter case, Z3 runs out of time on most of the benchmarks verifying 10 programs (3 safe; 7 unsafe) while SPACER can verify 97 programs (21 safe; 76 unsafe). We should mention that of the 7 unsafe programs verified by Z3, 5 could not be verified by SPACER.

In summary, we believe SPACER is a valuable addition to the state-of-the-art as shown above by its practical advantage on some hard device driver benchmarks.

VII. Related Work

VIII. Conclusion and Future Work

References

[1] "Software Verification Competition," TACAS, 2015, http://sv-comp.sosy-lab.org/.

[2] W. Ackermann, *Solvable Cases of The Decision Problem*. North-Holland, Amsterdam, 1954.

[3] A. Albarghouthi, A. Gurfinkel, and M. Chechik, "From Under-Approximations to Over-Approximations and Back," in *TACAS*, 2012.

[4] ——, "Whale: An Interpolation-Based Algorithm for Inter-procedural Verification," in *VMCAI*, 2012.

[5] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic Model Checking without BDDs," in *TACAS*, 1999.

[6] N. Bjørner, K. McMillan, and A. Rybalchenko, "Program Verification as Satisfiability Modulo Theories," in *SMT*, 2012.

[7] A. R. Bradley, "SAT-Based Model Checking without Unrolling," in *VMCAI*, 2011.

[8] W. Craig, "Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory," *Symbolic Logic*, vol. 22(3), 1957.

[9] L. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *TACAS*, 2008.

[10] N. Eén, A. Mishchenko, and R. K. Brayton, "Efficient Implementation of Property Directed Reachability," in *FMCAD*, 2011.

[11] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko, "Synthesizing Software Verifiers from Proof Rules," in *PLDI*, 2012.

[12] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. Navas, "The SeaHorn Verification Framework," in *CAV*, 2015.

[13] A. Gurfinkel, T. Kahsai, and J. A. Navas, "SeaHorn: A Framework For Verifying C Programs - (Competition Contribution)," in *TACAS*, 2015.

[14] M. Heizmann, J. Christ, D. Dietsch, E. Ermis, J. Hoenicke, M. Lindenmann, A. Nutz, C. Schilling, and A. Podelski, "Ultimate Automizer with SMTInterpol - (Competition Contribution)," in *TACAS*, 2013.

[15] K. Hoder and N. Bjørner, "Generalized Property Directed Reachability," in *SAT*, 2012.

[16] A. Komuravelli, A. Gurfinkel, and S. Chaki, "SMT-Based Model Checking for Recursive Programs," in *CAV*, 2014.

[17] K. L. McMillan and A. Rybalchenko, "Solving Constrained Horn Clauses using Interpolation," Microsoft Research, Tech. Rep. MSR-TR-2013-6, 2013.

[18] A. Stump, C. W. Barrett, D. L. Dill, and J. R. Levitt, "A Decision Procedure for an Extensional Theory of Arrays," in *LICS*, 2001.